Analyzing Class and Crosscutting Modularity with Design Structure Matrixes

Márcio de Medeiros Ribeiro[1], Marcos Dósea[1], Rodrigo Bonifácio[1], Alberto Costa Neto[1], Paulo Borba[1], Sérgio Soares[2]

[1] Informatics Center – Federal University of Pernambuco
Caixa Postal 7851, 50740-540 – Recife – PE – Brazil

[2]Computing Systems Department – Pernambuco State University
Rua Benfica, 455, Madalena, 50720-001 – Recife – PE – Brazil

{mmr3,mbd2,rba2,acn,phmb}@cin.ufpe.br, sergio@dsc.upe.br

Abstract. *Modularization of crosscutting concerns is the main benefit provided by Aspect-Oriented constructs. However, it does not address class modularity adequately. In order to assess both class and crosscutting modularity of AO systems, we use Design Structure Matrixes (DSMs) to analyze three different versions (OO, AO, and AO using design rules) of a real software application. We observed that, in the last version, coupling between classes and aspects is reduced, yielding a more modular design, specially when considering semantic dependencies between them. In addition, we apply new design parameters that represent a more realistic software development process.*

1. Introduction

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] has been proposed as a technique for modularizing crosscutting concerns. Logging, distribution, tracing, security, and transactional management are accepted as examples of crosscutting concerns well addressed by AOP. However, since AOP is a relatively new approach, there is not yet consensus about how to evaluate designs or even about which dimensions of modularity are supported by AOP.

In fact, by referring to classes implementation details in aspects, one can inhibit modular reasoning, requiring class modifications to be fully aware of the aspects affecting the class. Therefore, constructs aimed to support *crosscutting modularity* might actually break *class modularity*. In the presence of aspects, class modularity is compromised because, when evolving a class, it might be necessary to analyze the implementation of existing aspects, instead of analyzing only the class and the interface of other referred classes.

Tools like AJDT [AJDT 2007] help on identifying which aspects affect a class by putting marks in the code. Nevertheless, even in this case the developer needs to understand these aspects implementations because they can change the expected behavior of the class.

In order to solve the aforementioned modularity problems when considering AO systems, we argue that using a new category of design rules [Baldwin and Clark 2000] to decouple classes and aspects is essential. Design rules are not just guidelines and

recommendations: they generalize the notion of information hiding interfaces and must be rigorously obeyed.

In this paper, we follow similar approaches already adopted by others works [Sullivan et al. 2005, Lopes and Bajracharya 2006], but with two main differences: first, besides to software components, we introduce architectural style and use cases as design parameters, representing a more realistic software development process. Second, we consider a new kind of dependency between aspects and classes. Such dependency, named Semantic Dependencies [Neto et al. 2007], has a significant impact on dimensions of modularity, such as parallel development of modules.

Therefore, we provide in this work an approach more precise for analyzing AO software modularity, being useful for assessing which are the real benefits of AO concerning both class and crosscutting modularity.

Aiming at confirming this hypothesis, we have analyzed the structure of three versions of the Health Watcher (HW) system [Soares et al. 2002, Greenwood et al. 2007] using Design Structure Matrixes (DSMs). The OO version had problems to modularize crosscutting concerns. In order to solve this problem, an AO version was created. However, we observed that this approach reduces class modularity. In order to provide both class and crosscutting modularity, we analyze an AO version using design rules as well. Comparisons among such versions are detailed in this paper.

The main contributions of this paper are:

- Comparing and discussing modularity concepts presented in three versions (OO, AO, and AO with design rules) of a real software application. We observed that the AO version decreases the class modularity. Also, in order to obtain both class and crosscutting modularity, we argue that design rules must be used (Section 4).
- Applying new design parameters on assessing modularity through DSMs. Such parameters might represent a more realistic software development process, detailing responsibilities for developers. (Section 4).
- The necessity of applying semantic dependencies for modularity analysis between classes and aspects. We argue that those dependencies also should be expressed as design rules, reducing the dependencies between modules and, consequently, improving modularity (Section 3).

2. Background

In this section we present definitions related to modularity and design structure matrixes, an approach to evaluate the modularity of a complex design.

2.1. Modularity

The concept of modularity applied to software development was first introduced by Parnas [Parnas 1972]. Such concept is still used as a guide for architects and is being applied in another areas. Modularity is closely related to design decisions that decompose and organize the system into a set of modules. The following qualities attributes are expected in a modular design:

Comprehensibility A modular design allows developers to understand a module looking only at: (1) the implementation of the module itself; and (2) the interfaces of the other modules referenced by it[1].

Changeability A modular design enables local changes. If changes are necessary in the internal implementation of a module *A*, the other modules that depend exclusively on *A's interface* will not need to change, since there is no modification in the module interface.

Parallel development After the specification of the module interfaces, a modular design enables the parallel development of modules. Different teams might only focus in their own modules development, reducing the time-to-market and the need of communication.

Parnas proposed the *information hiding* principle as the criteria to be used in decomposition of systems into modules. According to Parnas, the parts of a system that are more likely to changes must be hidden into modules with stable interfaces.

The Object-Oriented approach enforces this decomposition criteria with directives to hide the implementation details of classes, exposing only their interfaces. However, the design of OO applications usually results in tangling and scattering code, due to crosscutting concerns implementation, reducing the degrees of comprehensibility and changeability.

Aspect-Oriented Programming was proposed to modularize these crosscutting concerns. However, constructions supported by AspectJ [Kiczales et al. 2001] like languages can produce high coupling between the base code and the aspects, because they are usually dependent on classes implementation details.

Baldwin and Clark [Baldwin and Clark 2000] proposed a theory which considers modularity as a key factor to innovation and market growth, independent of the industry area. Their theory uses Design Structure Matrixes (DSMs) to reason about dependencies among artifacts and defend that the task structure organization is closely related to them. In this way, if two modules are coupled, their parallel and independent development is impossible, requiring more communication between the different teams, or their implementation by a single team.

2.2. DSMs, Design Parameters, and Design Rules

Design Structure Matrixes (DSMs)[2] are used to visualize dependencies among *design parameters*. These parameters correspond to any decision that needs to be made along the product design.

Design parameters may have different abstraction levels. In software industry, some design decisions are related to process development, language, code/architectural style, and so forth. Moreover, if we consider implementation as design activities, software components like classes, interfaces, packages, and aspects should be represented as design parameters.

The notion of dependency arises whenever a design decision depends on another. Each design parameter is disposed in both rows and columns of the matrix. A dependency

[1]This comprehensibility degree is also known as *modular reasoning*.

[2]*Dependency Structure Matrix* is another term used.

between two parameters is marked with a X.

Figure 1 represents software components as parameters in a DSM. A mark in row B, column A represents that component B depends on component A. In the same way a X in row A, column B represents that component A depends on component B. Whenever this mutual dependency occurs, we have an example of *cyclical dependency*, which implies that both components can not be independently addressed, which means that their parallel development is compromised.

	A	B	C
A		X	
B	X		X
C			

Figure 1. Example of dependencies in a DSM.

Additionally, component B depends on C (expressed by a X in row B, column C) but C does not depend on any other component. Therefore, C can be independently developed but B can not be completely developed until C has been concluded. Aiming at removing these dependencies, Baldwin and Clark [Baldwin and Clark 2000] propose defining Design Rules.

Design Rules are parameters used as interfaces between modules that are less likely to be changed [Lopes and Bajracharya 2006]. In this way, they can promote decoupling of design parameters, like interfaces decrease the coupling between software components. Such design rules establish strict partitions of knowledge and effort at the outset of a design process. They are not just guidelines or recommendations: they must be rigorously obeyed in all phases of design and production [Baldwin and Clark 2000].

3. Semantic Dependencies

In our previous work [Neto et al. 2007], we have presented an approach of modularity analysis that considers both syntactic and semantic dependencies between classes and aspects. In what follows, we discuss more deeply such dependencies, using a detailed example.

Syntactic dependency in OO software components (classes and interfaces) occurs when there is a direct reference between them, such as inheritance, composition, methods signatures (parameters, return types, exceptions throwing), class instantiations, and so forth. This dependency causes compile errors whenever a component is modified or removed from the system, being thus easily detected. In the same way of classes and interfaces, direct references can appear between aspects and other components, which means that aspects can also have syntactic dependency.

However, there is another kind of dependency that is not so easy to realize because it occurs without explicit references between system components (classes, interfaces, and aspects - in AO systems). We call this kind of coupling as *Semantic* dependency. Besides, this kind of dependency does not cause compilation errors when removing or modifying components. Although there are semantic dependencies in OO systems, as occurs for example when using reflection to call methods from a class (in this case it is not necessary using identifiers explicitly), we did not find semantic dependencies in the OO HW system.

As an example of semantic dependency in AO systems, suppose the requirement of synchronizing all methods of a class (concurrency management). Such requirement consists of encompassing with synchronized blocks the bodies of all methods. Consider that the aspect developer is responsible for implementing this concern in an aspect and that a class developer, which is oblivious about this aspect, decides to implement methods in the same class. In this situation, at least three problems can occur:

1. The methods created by the class developer might not need concurrency management, but they will be synchronized by the aspect;
2. If the class developer, oblivious about the aspect, implements concurrency management on methods class, these methods would be synchronized twice; and
3. Depending on how the synchronization approaches are implemented by the aspect and class developers, together they might lead the system to a dealock or a livelock situation [Lea 1999].

In such cases, the expected behavior of the system could be compromised, since some additional synchronization would be created or, even worst, the system might reach a dealock or a livelock.

The situation above exposes that problems of modularity have occurred: (1) the comprehensibility is compromised, since two modules should be studied in order to understand the concern; and (2) the parallel development is problematic, because one developer can implement unintended behavior into a module which, although it is not under his responsibility, might break the system.

These problems are caused by semantic dependencies, because the class depends on the aspect to work correctly. They occur since the class developer is oblivious about the aspect implementation (there is no syntactic definition that the concurrency concern is woven in the class by the aspect). Such problem can also appear when, for example, the aspect developer changes the concurrency implementation. In this case, the aspect developer might break the expected behavior of the class. We have observed the same kind of dependency in other scenarios in the HW system, like transactional management and distribution.

In this paper, we propose using design rules in order to reduce not only syntactic but also semantic dependencies between aspects and classes. Using design rules requires both aspects and classes developers agreement. In this way, it promotes the decoupling between such components, promoting modularity.

Figure 2 illustrates the development using design rules between classes and aspects. The design rule makes explicit that every method in the *EmployeeRepository* class will be synchronized by the *Concurrency* aspect and that the *EmployeeRepository* class can not implement any synchronization mechanism.

Hence, the class developer is not oblivious anymore when considering the implementation of the concurrency concern. The design rule will constrain the developer while creating methods in the *EmployeeRepository* class, eliminating the semantic dependencies between the aspect and the class. If the methods must be synchronized, the developer just create them and leave the concurrency responsibility for the aspect. Otherwise, he must communicate to the aspect developer that some methods will not need concurrency management. This is the same expected approach when developing OO

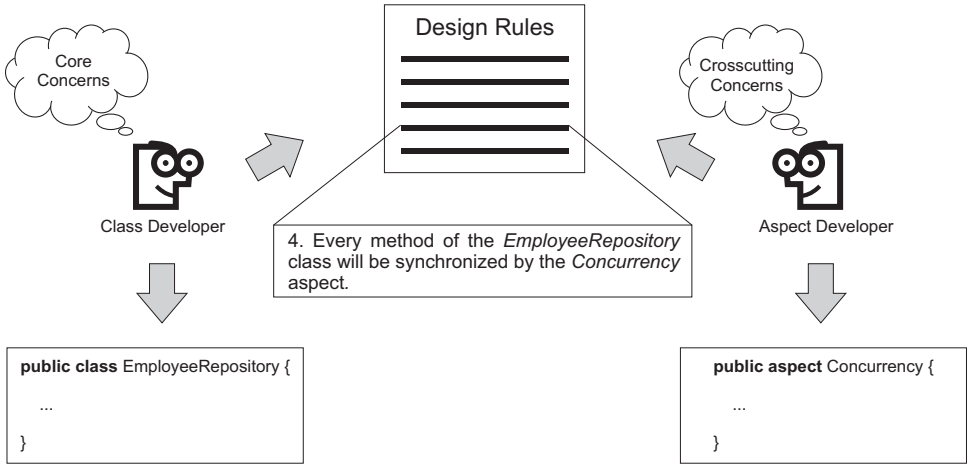

Figure 2. Aspect and Class developers using Design Rules.

systems using design mechanisms like Java interfaces, and the Design by Contract approach [Meyer 1992, Meyer 1997].

4. Modularity Analysis

This section describes the analysis of modularity of three different versions of the Health Watcher system, a real web-based information system, originally implemented in Java and restructured to use AspectJ [Kiczales et al. 2001], a general purpose AO extension to Java. The system aims to improve the quality of the services provided by health care institutions, allowing citizens to register complaints regarding health issues, and health care institutions to investigate and take the required actions. This system was selected because it was used in many previous works [Soares et al. 2002, Greenwood et al. 2007, Neto et al. 2007] and its design has a significant number of non-crosscutting and crosscutting concerns. Furthermore, it requires a number of common day-to-day design decisions related to GUI, persistence, transaction management, distribution, and concurrency control.

Following an use case driven approach, we choose to represent four categories of design parameters: constraints and requirements, use cases, architectural decisions, and software components (classes and aspects). Two kinds of DSMs are presented in the remainder of this section. The first one represents a high level view of the HW design parameters, whereas the second one offers a detailed view of design parameters related to the Employee component[3]. Not all requirements are represented in the second one. The comparative analysis are based on both kinds of matrixes.

Since the DSMs were constructed based on an use case driven approach (an important differential of our work), it is possible to infer the organization and dependencies between the tasks necessary to develop the system from the dependencies between artifacts. Because the tasks can be obtained from the DSM, it is easier to separate teams to perform such tasks, by identifying which tasks can be or not developed in parallel.

[3]Similar matrixes could be derived for other components, like Health Unit, Complaint, and Authentication.

4.1. OO Health Watcher version

Figure 3 illustrates the general parameter organization of the OO Health Watcher version. The DSM is divided in four groups of design parameters:

- **Constraints and Requirements**: related to system *constraints and requirements*, identified in the beginning interactions.
- **Use Cases**: represent system use cases. They are categorized as *architectural*, *application*, and *utility* use cases.
- **Architectural Decisions**: guide the project development, consisting of design rules that are related to: architectural style, patterns, frameworks, and Application Programming Interfaces (APIs). The establishment of a system architecture depends on the development team experience and the system requirements, constraints, and architectural use cases.
- **Components**: consist of software *components* that implement the use cases previously defined.

	Design Parameters		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	Constraints and Requirements	1																					
Use Cases	Login	2	x	Architectural																			
	Register new employee	3	x																				
	Register new complaint	4	x																				
	Query information	5	x				Application																
	Update employee	6	x																				
	Update complaint	7	x																				
	Update health unit	8	x																				
	Register tables	9	x								Utility												
	Change logged employee	10	x																				
Architectural Decisions	Select architectural style and patterns	11	x	x	x	x							x										
	GUI technology	12	x	x	x	x							x										
	Security mechanism	13	x	x	x	x							x										
	Persistence	14	x	x	x	x							x										
	Distribution mechanism	15	x	x	x	x							x										
	Concurrency mechanism	16	x	x	x	x							x										
	Transaction mechanism	17	x	x	x	x							x										
Components	Employee	18			x		x	x					x	x	x	x	x	x	x				
	Health Unit	19					x			x	x		x	x	x	x	x		x				
	Complaint	20				x	x		x				x	x	x	x	x	x	x	x	x		
	Authentication	21		x								x	x	x	x	x	x		x	x			

Figure 3. Design structure of the OO Health Watcher version.

Figure 3 illustrates that the *use case* specifications depend on the previously identification of system *constraints and requirements* (rows 2-10, column 1).

Another existing example of dependency occurs between *Complaint component* and *Register New Complaint*, *Query Information*, and *Update Complaint* use cases. Such dependencies are marked in row 20, columns 4, 5, and 7. The component also depends on the *architectural style* (row 20, column 11). In addition, it depends on all mechanisms that implement concerns such as *Transaction* and *Concurrency* management (row 20, columns 16 and 17). Finally, the *Complaint component* depends on the *Employee* and *Health Unit* components (row 20, columns 18 and 19), since it is possible to reuse part of responsibilities associated with them.

Figure 4 details the specific parameters of the *Employee* component (row 18 of Figure 3). The team allocated to implement such component must develop specific responsibilities for the *Register New Employee*, *Update Employee*, and *Query Information*

use cases. Also, they must implement responsibilities that are spread throughout the system, such as *Distribution*, *Concurrency*, and *Transaction* management concerns (these dependencies are represented in row 18, columns 12, 13, and 14, respectively).

Design Parameters			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	Constraints and Requirements	1																			
Use Case	Register new employee	2	x																		
	Update employee	3	x																		
	Query information	4	x																		
Architectural Decisions	Select architectural style and patterns	5	x	x																	
	Distribution mechanism	6	x	x																	
	Concurrency mechanism	7	x	x																	
	Transactional mechanism	8	x	x																	
	Persistence mechanism	9	x	x																	
	Security mechanism	10	x	x																	
	GUI technology	11	x	x																	
Employee	Distribution concern	12		x			x	x													
	Concurrency concern	13		x			x		x												
	Transaction management concern	14		x			x			x											
	Persistence Interface	15		x	x	x	x				x										
	Persistence Impl	16		x	x	x	x				x						x			x	
	Security	17		x	x	x	x					x									
	Business	18		x	x	x	x							x	x	x	x				
	GUI	19		x	x	x	x						x	x					x	x	

Figure 4. Object-Oriented Employee component details.

With respect to parallel development, it is not feasible because of the lack of crosscutting modularity. Moreover, changes are not localized: if the *Distribution* mechanism changes, it will be necessary to update the *GUI* and *Business* design decisions not only in the *Employee* component, but also in other components such as *Health Unit*, *Complaint*, and *Authentication*.

In summary, despite of the OO version presents class modularity (as expected), it does not modularize adequately the crosscutting concerns, like *Distribution*, *Concurrency* (illustrated in Listing 1 by the presence of many synchronized blocks within *Business* class *EmployeeRepository*) and *Transaction* management, since such concerns are tangled and scattered in such component.

Listing 1. Concurrency concern tangled and scattered in EmployeeRepository.

```
1  public class EmployeeRepository {
2
3      public void insert(Employee employee) {
4          synchronized (this) { // Business logic to insert Employees }
5      }
6
7      public Employee search(String login) {
8          synchronized (this) { // Business logic to search Employees }
9      }
10
11     public void update(Employee employee) {
12         synchronized (this) { // Business logic to update Employees }
13     }
14
15     public void remove(Employee employee) {
16         synchronized (this) { // Business logic to remove Employees }
17     }
18 }
```

4.2. AO Health Watcher version

Figure 5 illustrates the AO Health Watcher design structure. This matrix is similar to the OO version when considering requirements, constraints, and use cases. The architectural decisions also present the same set of OO parameters.

Additionally, in the same figure, the *Employee*, *Health Unit*, *Complaint*, and *Authentication* components do not syntactically depend on crosscutting mechanisms. Notice that such dependencies were encapsulated in the aspects responsible for implementing the *Distribution*, *Concurrency* and *Transaction* management (rows 22, 23, and 24; columns 15, 16, and 17). These aspects also have dependencies with the application components.

	Design Parameters		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	Constraints and Requirements	1																								
Use Cases	Login	2	x	Architectural																						
	Register new employee	3	x																							
	Register new complaint	4	x																							
	Query information	5	x				Application																			
	Update employee	6	x																							
	Update complaint	7	x																							
	Update health unit	8	x																							
	Register tables	9	x								Utility															
	Change logged employee	10	x																							
Architectural Decisions	Select architectural style and patterns	11	x	x	x	x																				
	GUI technology	12	x	x	x	x							x													
	Security mechanism	13	x	x	x	x							x													
	Persistence	14	x	x	x	x							x													
	Distribution mechanism	15	x	x	x	x							x													
	Concurrency mechanism	16	x	x	x	x							x													
	Transaction mechanism	17	x	x	x	x							x													
Components	Employee	18			x		x	x					x	x	x	x										
	Health Unit	19					x			x	x		x	x	x	x										
	Complaint	20				x	x		x				x	x	x	x				x	x					
	Authentication	21		x								x	x	x	x	x				x						
AOP	Distribution aspect	22											x				x			x	x	x	x			
	Concurrency aspect	23											x					x		x		x				
	Transaction aspect	24											x						x	x	x	x	x			

Figure 5. Aspect-Oriented Health Watcher design structure.

Figure 6 presents details about the *Employee* component in the AO version. Such component does not have parameters related to *Distribution*, *Concurrency* and *Transaction* management (as in the OO version in Figure 4). Notice that these concerns (implemented as aspects, rows 12, 13, and 14 of Figure 6) are not only used in the *Employee* component, but also in the *Health Unit*, *Complaint*, and *Authentication*[4] components.

With respect to modularity, the *Business* component became independent of crosscutting concerns implementations (row 18, columns 12, 13, and 14). Besides, changes are more localized in this design. For example: if the *Distribution* mechanism changes, it will not be necessary to update the *GUI* and *Business* components (there is no mark in rows 18 and 19 against column 12).

Listing 2 illustrates the aspect responsible for modularizing the *Concurrency* concern, which eliminates both tangled and scattered synchronized blocks from the original *EmployeeRepository* class (Listing 1).

Listing 2. Concurrency concern tangled and scattered in EmployeeRepository.

```
1  public aspect LocalSynchronization {
2
3      Object around(Object o): this(o) && execution(* EmployeeRepository.*(..)) {
4          synchronized(o) {
5              return proceed(o);
6          }
7      }
8  }
```

[4]Although *Authentication* is a crosscutting concern, it is not implemented using AOP in the HW.

	Design Parameters		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	Constraints and Requirements	1																			
Use Case	Register new employee	2	x																		
	Update employee	3	x																		
	Query information	4	x																		
Architectural Decisions	Select architectural style and patterns	5	x	x																	
	Distribution mechanism	6	x	x																	
	Concurrency mechanism	7	x	x																	
	Transactional mechanism	8	x	x																	
	Persistence mechanism	9	x	x																	
	Security mechanism	10	x	x																	
	GUI technology	11	x	x																	
AOP	AO Distribution aspect	12		x			x	x												x	x
	AO Concurrency aspect	13		x			x		x											x	
	AO Transaction management	14		x			x			x										x	
Employee	Persistence Interface	15		x	x	x	x				x										
	Persistence Impl	16		x	x	x	x				x						x			x	
	Security	17		x	x	x	x					x									
	Business	18		x	x	x	x										x				
	GUI	19		x	x	x	x						x						x	x	

Figure 6. Aspect Oriented Employee component details.

4.3. AO Health Watcher version with Semantic Dependencies

Existing works about modularity consider only syntactic dependencies [Zhao 2004, Santánna et al. 2003, Garcia et al. 2006], ignoring the existence of semantic dependencies and drawing similar conclusions to the presented in the previous section.

The application components do not have syntactic references to the aspects. However, situations like the concurrency management (explained in Section 3) expose the existence of semantic dependencies.

The semantic dependencies of the AO HW version are represented in Figure 7 as plus symbols (+), showing that the *Employee*, *Health Unit*, *Complaint*, and *Authentication* components depend on the aspects that implement the crosscutting concerns. In this way, when evolving classes it is necessary to analyze the aspects, as described in the following scenario.

Suppose a new feature that requires counting the number of Employees registered in a given profile. Such feature might be implemented as a new method in the *EmployeeRepository* class (Listing 3). This method does not need to be synchronized, since it does not manipulate a specific instance of *Employee*. However, the application will not behave as presumed, since the method will be synchronized by the aspect.

Listing 3. A new method for counting the number of Employees.

```
1 public class EmployeeRepository {
2
3     public int getNumberOfEmployees(Profile profile) {
4         // Business logic to count the number of Employees
5     }
6
7 }
```

As expected, the AO version presented a better crosscutting modularity. However, we observed problems related to class modularity as a result of aspect-class compositions, since it is not possible to reason about each class separately (without observing all existing aspects).

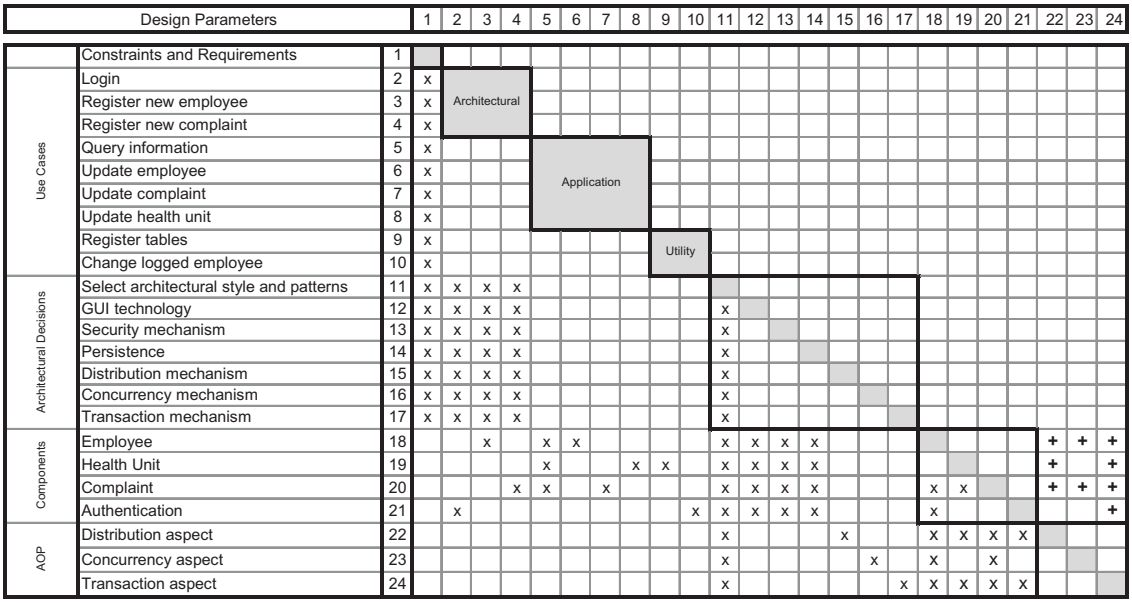

Figure 7. Aspect Oriented DSM with Semantic Dependencies.

4.4. AO Health Watcher version with New Design Rules

Design Rules in AO systems must include additional constraints to define how classes and aspects interact. Examples of such constraints are: required join point, name patterns, class members, general invariants, and pre/post-conditions.

The definition of design rules that decouple classes and aspects is a non-trivial task. This activity depends on previous experiences in the development of applications with the same architectural characteristics. One possible solution is the establishment of design rules during the architectural use case development activity. In the absence of the aforementioned design rules, both class and aspect developers must be aware of each other and communicate frequently to state how classes and aspect interact, which would not allow the parallel development.

For example, in order to avoid the problem of unintended synchronization (presented in Section 4.3), we could recur to the design rule defined in Figure 2 to eliminate the semantic dependency between *EmployeeRepository* and the aspect responsible for implementing the *Concurrency* concern. Similar design rules were defined to decouple classes from *Distribution* and *Transaction* aspects.

Figure 8 presents a new parameter (row 18) corresponding to the design rules that decouple classes and aspects. Notice that using these design rules the directly dependencies between classes and aspects were reduced and localized in the *New Design Rules* (dependencies marked in the intersection of rows 19-25 with column 18).

By introducing design rules, which play the role of interfaces between classes and aspects, classes must only be aware of the design rules they depend on, ignoring all aspects implementations, achieving both class and crosscutting modularity. In summary, we conclude that design rules are a useful technique to construct modular AO systems.

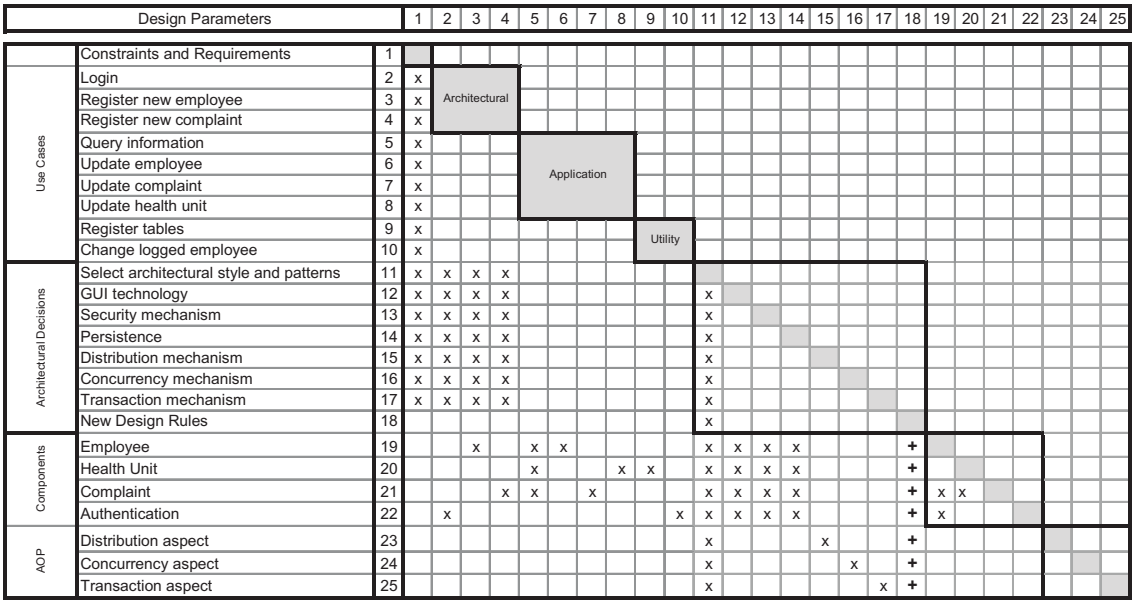

Figure 8. Aspect Oriented DSM with New Design Rules.

5. Related Work

Sullivan *et al.* [Sullivan et al. 2005] presented a comparative analysis, also based on DSMs, between an AO system developed following an oblivious approach, with the same system developed with clear design rules that document interfaces between classes and aspects. Such work does not consider parameters like architectural style and use cases as we do. Griswold *et al.* [Griswold et al. 2006] showed how to transform part of the design rules into a set of Crosscutting Programming Interfaces (XPIs) that are useful to document and check part of the design rules (contracts). In contrast, these works do not consider semantic dependencies, leading to unfaithful notion of dependencies when considering AO systems. In our work, the design rules are responsible for dealing with dependencies (both syntactic and semantic) between classes and aspects. They could be mapped into XPIs, but we are convinced that, even without defining these XPIs, they are of great importance to achieve a better modularity in AO development.

It is important to notice that we considered crosscutting concerns like Transaction Management, Concurrency, Persistence, and Distribution, that are present in most of applications nowadays, drawing our conclusions over real problems faced by OO and AO developers, instead of concentrating in analyzing concerns that are usually subsidiary features, like Logging and Tracing.

Kiczales and Mezini [Kiczales and Mezini 2005] defend that the complete interface of a module can only be determined once the complete configuration of the systems is known. They introduce the notion of aspect-aware interfaces, that describe the existing dependencies between classes and aspects, giving support to reasoning about the effect of aspects over classes by point out where aspects are affecting a certain class. This interface must be automatically recomputed whenever classes or aspects change and, in fact, tools like AJDT already offer similar functionality through IDE resources that indicate which advices apply in a certain point. However, we do not believe that this kind of interface can really help during earlier phases of software development, when it is necessary to parti-

tion the system into classes and aspects, and distribute the task developing them between teams. At that moment, it is necessary to establish design rules that will govern both class and aspects development.

Lopes and Bajracharya [Lopes and Bajracharya 2006] used DSM and Net Option Value (NOV) to compare the value of modularity achieved by different design options. They concluded that aspects can increase the value of an already modularized design. We did not measure the HW NOV but we discussed the dependencies observed in the DSMs, both in OO and AO versions, and reason about their influence over parallel development, flexibility, and comprehensibility. In addition, we explored the concept of dependencies between aspects and classes in more detail and confirmed the importance of establishing design rules as a way of dealing with these dependencies.

6. Concluding Remarks

We have shown in this work a more precise study for analyzing software modularity, being useful for assessing what are the real benefits of AO concerning both class and crosscutting modularity.

Additionally, we discussed that Crosscutting Concerns can be localized in single modules (aspects), providing better crosscutting modularity. On the other hand, this approach does not provide class modularity because in order to reason about any class it is necessary to consider all aspects implementations.

Aiming to confirm that, we have presented an analysis of different versions of the Health Watcher system using DSMs. We shown that the AO version provides crosscutting modularity but at the same time reduce class modularity. This weakness can be mitigated by using adequate design rules between classes and aspects.

Such design rules were necessary in order to reduce syntactic and semantic dependencies in the AO version. Another version considering design rules was analyzed and we confirm that it provides a better design for modularity than the other ones.

Finally, we showed the possibility of using different parameters in DSMs. Such parameters consist of not only software components as used in other works, but also architectural decisions and use cases, representing a more realistic software development process.

7. Acknowledgments

We would like to thank CNPq and CAPES, Brazilian research funding agencies, for partially supporting this work. In addition, we thank SPG[5] members for feedback and fruitful discussions about this paper.

References

AJDT (2007). Getting started with AJDT. http://www.eclipse.org/ajdt/gettingstarted.php.

Baldwin, C. Y. and Clark, K. B. (2000). *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press.

[5]http://www.cin.ufpe.br/spg

Garcia, A., SantÁnna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A. (2006). Modularizing Design Patterns with Aspects: A Quantitative Study. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 36–74. Springer.

Greenwood, P., Bartolomei, T., Figueiredo, E., Dósea, M., Garcia, A., Cacho, N., SantÁnna, C., Soares, S., Borba, P., Kulesza, U., and Rashid, A. (2007). On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proceedings of the 21st European Conference on Object–Oriented Programming, ECOOP'07*. to appear.

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. (2006). Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. (1997). Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242.

Kiczales, G. and Mezini, M. (2005). Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 49–58. ACM Press.

Lea, D. (1999). *Concurrent Programming in Java*. Addison–Wesley, 2nd edition.

Lopes, C. V. and Bajracharya, S. K. (2006). Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 1–35. Springer.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition.

Neto, A. C., de Medeiros Ribeiro, M., Dósea, M., Bonifácio, R., Borba, P., and Soares, S. (2007). Semantic Dependencies and Modularity of Aspect-Oriented Software. In *1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering (ICSE'07)*. to appear.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Santánna, C., Garcia, A., Chavez, C., Lucena, C., and von Staa, A. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: A Assessment Framework. In *Proc. of Brazilian Symposium on Software Engineering (SBES'03)*, pages 19–34.

Soares, S., Laureano, E., and Borba, P. (2002). Implementing distribution and persistence aspects with AspectJ. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'2002*, pages 174–190, Seattle, USA.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of*

the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), pages 166–175, New York, NY, USA. ACM Press.

Zhao, J. (2004). Measuring Coupling in Aspect-Oriented Systems. In *10th International Software Metrics Symposium (Metrics'04)*.